DP1 2020-2021

Documento de Diseño del Sistema

Proyecto MADAJA

https://github.com/gii-is-DP1/dp1-2020-g1-05

Miembros:

- Echegoyán Delgado, Álvaro
- Pérez Carrillo, Manuel
- Pérez Plata, Juan José
- Pérez Vázquez, Antonio
- Piury Pinzón, Alejandro
- Toro Valle, Daniel

Tutor: Irene Bedilia Estrada Torres

GRUPO G1-05

Versión 1.0

10/01/2021

Historial de versiones

Fecha	Versión	Descripción de los cambios	Sprint
03/01/2021	V1	• Creación del documento	3
09/01/2021	V2	• Añadido diagrama de dominio/diseño • Añadido diagrama de capas • Explicación de la aplicación del patrón MVC • Explicación de las decisiones de diseño	3
03/02/2021	V3	• Añadido nuevas decisiones de diseño • Añadido nuevos patrones y estilos	4
09/02/2021	V4 (Final)	• Añadido últimas decisiones de diseño • Añadido diagrama de capas dividido • Añadido diagramas de dominio separados	4

Índice

1. Introducción

En este proyecto vamos a desarrollar un sistema de información que se encargue de gestionar correctamente una empresa dedicada al alquiler y venta de vehículos, tanto en establecimientos físicos como de manera online.

Su funcionalidad principal es la de exponer un catálogo de vehículos disponibles, con sus ofertas, por esta empresa para que los clientes puedan realizar y gestionar adecuadamente sus compras y alquileres de manera online, sin necesidad de desplazarse hasta un concesionario físico. También les permite solicitar el envío de un vehículo hasta la localización que deseen además de gestionar los seguros.

No solo ofrece estas facilidades para los clientes, sino que también permite una mejor gestión a nivel interno (para el administrador y para los trabajadores de la empresa) de todos los datos que se manejan.

El objetivo de este sistema es que cumpla todos los requisitos especificados además de que sea fácil de entender y cómodo de usar tanto para clientes como para trabajadores.

2. Diagramas UML

2.1. Diagrama de Dominio/Diseño

Ver Anexo I. Diagrama de Dominio/Diseño.

2.2. Diagrama de Capas

Ver Anexo II. Diagrama de Capas.

3. Patrones de diseño y arquitectónicos aplicados

En esta sección de especificar el conjunto de patrones de diseño y arquitectónicos aplicados durante el proyecto. Para especificar la aplicación de cada patrón puede usar la siguiente plantilla:

Patrón: Single Page Application(SPA)

Tipo: Arquitectónico

Contexto de Aplicación

Este patrón se ha aplicado en todas las vistas de la API, usando AJAX en todos los scripts de JavaScript con ruta: /dp1-2020-g1-05/src/main/resources/static/resources/js. Las vistas afectadas son: mostrarMisAlquileresAPI.jsp, concesionarioDetailsAPI.jsp, mostrarConcesionariosAPI.jsp, createOfertaFormAPI.jsp, mostrarOfertasAPI.jsp y ofertaDetailsAPI.jsp, updateOfertaAPI.jsp.

Clases o paquetes creados

Los scripts creados para llevar a cabo este patrón son: concesionarioDetails.js, createOfertaAPI.js, mostrarConcesionarios.jsp, mostrarMisAlquileres.jsp, mostrarOfertas.jsp, ofertaDetails.jsp, updateOfertaAPI.jsp.

Ventajas alcanzadas al aplicar el patrón

Este patrón lo hemos usado para integrar la API, puesto que constituye una buena forma de integración con los distintos microservicios, es fácil de debuguear y detectar errores de código, una interfaz sencilla y construcción del HTML dinámico y una respuesta fluida.

Patrón: Identity Field

Tipo: Diseño

Contexto de Aplicación

Este patrón se ha aplicado en todas las entidades y actores del sistema, estableciendo un número de identificación en cada una de ellas. La ruta o paquetes donde se ha aplicado este patrón es: com/springframework/samples/madaja/model.

Clases o paquetes creados

En este caso no has sido necesario crear clases o paquetes para implementar este patrón puesto que hemos identificado cada entidad con un atributo etiquetándolo con @Id de javax persistence.

Ventajas alcanzadas al aplicar el patrón

Aplicar un identificador único a cada entidad es equivalente a la clave primaria en la base de datos, por lo que nos permite que cada entidad sea única y si se crea una nueva se autogenere su identificador único.

Patrón: Layer Supertype

Tipo: Diseño

Contexto de Aplicación

Este patrón se ha aplicado en el modelo, en las entidades identificadas. La ruta o paquete afectado es: com/springframework/samples/madaja/model.

Clases o paquetes creados

Para implementar este patrón ha sido necesario crear dos clases abstractas, BaseEntity.java y NamedEntity.java.

Ventajas alcanzadas al aplicar el patrón

Crear una clase abstracta para todas aquellas clases que necesiten un identificador único nos permite tener el código mas estructurado, comprensible y limpio, evitando tener que repetir en cada entidad la identificación autogenerada de su ID y de un atributo “name” en aquellas que lo necesiten.

Patrón: Repository Pattern

Tipo: Diseño

Contexto de Aplicación

Este patrón se ha aplicado en la capa de recursos, es decir en el paquete: com/springframework/samples/madaja/repository.

Clases o paquetes creados

Para implementar este patrón ha sido necesario crear todas las clases de repositorio contenidas en el paquete mencionado anteriormente.

Ventajas alcanzadas al aplicar el patrón

Los repositorios contienen toda la lógica requerida para acceder a los datos, por tanto nos facilita las operaciones a realizar con la base de datos, simulando la base de datos como una colección de datos en memoria.

Patrón: Domain Model

Tipo: Diseño

Contexto de Aplicación

Este patrón se ha aplicado a todas las entidades del modelo: com/springframework/samples/madaja/model.

Clases o paquetes creados

Para implementar este patrón ha sido necesario crear todas las entidades del dominio de negocio y establecer sus relaciones conforme al diagrama UML obtenido a partir de las historias de usuario.

Ventajas alcanzadas al aplicar el patrón

Las entidades representan objetos del dominio de negocio que contiene tanto datos como las relaciones, por lo tanto los objetos representan una tabla de la base de datos, lo que nos permite implementar lógica de negocio compleja.

Patrón: Service Layer

Tipo: Diseño

Contexto de Aplicación

Este patrón se ha aplicado a todas las clases del paquete de service: com/springframework/samples/madaja/service.

Clases o paquetes creados

Para implementar este patrón ha sido necesario crear todas las clases services contenidas en el paquete anteriormente mencionado.

Ventajas alcanzadas al aplicar el patrón

Usando este patrón, nos permite que los servicios de cada entidad contengan todas las operaciones permitidas con dicha entidad encapsulando tanto lógica de dominio como lógica de aplicación.

Patrón: Dependency Injection

Tipo: Diseño

Contexto de Aplicación

Este patrón se ha aplicado en todas las clases que dependía de otras, en el constructor de la clase, en general, las clases de servicio que necesitaban inyectar la dependencia de los repositorios, y las clases controladores que necesitaban inyectar la dependencia de los servicios.

Clases o paquetes creados

No ha sido necesario crear clase o paquetes para este patrón puesto que con solo con la etiqueta @Autowired inyectamos la dependencia.

Ventajas alcanzadas al aplicar el patrón

Nos aseguramos de que solo una instancia de esta clase es creada y de que las clases que necesitan de ésta puedan acceder a ella. Además, si se producen cambios en la clase nos evitamos tener que cambiar código para establecer la dependencia.

Patrón: The Front Controller Pattern

Tipo: Diseño

Contexto de Aplicación

Este patrón se ha aplicado en los controladores de la aplicación en general, en el paquete: com\springframework\samples\madaja\web

Clases o paquetes creados

No ha sido necesario crear clase o paquetes para este patrón puesto que spring nos proporciona la implementación del DispatcherServlet. Lo que sí hemos tenido que implementar son las etiquetas @GetMapping, @PostMapping, etc… en las clases controladores para identificar la respuesta HTTP del manejador de peticiones.

Ventajas alcanzadas al aplicar el patrón

Con este patrón proporcionado por el framework SPRING nos permite centralizar el control de las peticiones además el manejador de las peticiones se convierte en mucho más flexible y además este front controller de Spring no solo despacha las peticiones, sino que las procesa y las convierte proporcionando funcionalidades extras como validaciones o convertidores.

Patrón: MVC

Tipo: Diseño

Contexto de Aplicación

Este patrón se ha aplicado en los controladores y las vistas de la aplicación: Paquetes: com\springframework\samples\madaja\web

Clases o paquetes creados

Para este patrón no ha sido necesario crear clases específicas para dicho patrón, simplemente organizar la estructura y aprovecharla puesto que estamos usando Spring MVC. En los controladores, enviamos a la vista un ModelAndView de tal forma que en la vista podemos usar los datos proporcionados por el modelo, a la vez que el controlador responde de las diversas peticiones de la vista.

Ventajas alcanzadas al aplicar el patrón

Este patrón nos facilita en gran medida el trabajo de enviar los datos a la vista, pudiendo ser mucho más flexible esta operación y fácil de representar dichos datos a la vista. Además, favorece a los 3 principios de diseño básicos, alta cohesión, bajo acoplamiento y separación de responsabilidades.

Patrón: Template view

Tipo: Diseño

Contexto de Aplicación

Para las vistas de nuestro proyecto usamos JPS/JSTL, que usan el patrón Template view. Este patrón se ha aplicado en las vistas de la aplicación: Directorio: /dp1-2020-g1-05/src/main/webapp/WEB-INF/jsp

Clases o paquetes creados

No ha sido necesario crear clases específicas para este patrón. Solo usar las etiquetas que nos proporcian JSP y Spring.

Ventajas alcanzadas al aplicar el patrón

Las etiquetas de JSP son etiquetas HTML más algunas etiquetas especiales. Estas etiquetas especiales son bloques de construcción reutilizables para una página web. Spring también proporciona un conjunto de etiquetas que proporcionan funciones para facilitar el trabajo con las funciones de Spring.

Patrón: Eager/Lazy loading

Tipo: Diseño

Contexto de Aplicación

Este patrón se ha aplicado en todas las relaciones entre entidades del sistema.

Clases o paquetes creados

En este caso no has sido necesario crear clases o paquetes para implementar este patrón puesto que hemos marcado cada relación con las etiquetas @OneToOne, @ManyToOne, @OneToMany y @ManyToMany. Las dos primeras etiquetas usan Eager Loading por defecto y las dos últimas usan Lazy Loading.

Ventajas alcanzadas al aplicar el patrón

Es fácil de usar ya que solo hay que poner las etiquetas anteriores según las relaciones de nuestro modelo. Eager Loading permite que no haya impacto en el rendimiento retrasado, pero tiene tiempos de carga iniciales más lentos y puede cargar demasiados datos. Por otra parte, Lazy Loading tiene tiempos de cargas iniciales más rápidos y consume menos memoria, pero tiene impacto retrasado en el rendimiento y a veces, si la sesión expira, pueden surgir excepciones.

Patrón: Pagination

Tipo: Diseño

Contexto de Aplicación

Este patrón se ha aplicado en las entidades cliente, concesionario y vehiculos: com/springframework/samples/madaja/model.

Clases o paquetes creados

Para implementar este patrón no ha sido necesario crear ninguna clase, solo añadir el objeto Pageable como parámetro de entrada de los métodos que se quieren paginar.

Ventajas alcanzadas al aplicar el patrón

La paginación mejora la eficiencia al reducir el número datos de una consulta. También mejora la experiencia de usuario, al reducir el número de datos, estos tardarán menos en mostrarse.

Patrón: Strategy

Tipo: Diseño

Contexto de Aplicación

Este patrón ha sido utilizado para llevar a cabo las funcionalidades de seguridad del proyecto.

Clases o paquetes creados

Para implementar este patrón no ha sido necesario crear ninguna clase, solo añadir las urls que queríamos controlar a la cadena de filtros de la clase SegurityConfiguration.java.

Ventajas alcanzadas al aplicar el patrón

La seguridad implementada permite restringir el acceso a los usuarios que no tengan autoridad para realizar ciertas operaciones a cometer acciones maliciosas

4. Decisiones de diseño

4.1. Decisión 1: Importación de datos reales para paginaciones

Descripción del problema:

Nos gustaría poder introducir una gran cantidad de datos en el sistema para poder simular casos de uso realistas. El problema es incluir manualmente todos esos datos como parte del script de inicialización de la base de datos.

Alternativas de solución evaluadas:

Alternativa 1.a: Realizar funciones que generen automáticamente (en algunos casos usando una lista de datos externa) una gran cantidad de datos cómo NIF, nombres, listas de vehículos.

Ventajas:

- No requiere introducir manualmente el SQL que genere los datos.
- Más rápido que introducirlos manualmente

Inconvenientes:

- Ralentiza el desarrollo de funcionalidades principales del sistema.
- Tenemos que buscar nosotros listas con datos reales
- Bastante más complejo y tedioso

Alternativa 1.b: Introducir manualmente una cantidad reducida de datos en el SQL pero que sea suficiente para realizar paginaciones.

Ventajas:

- Podemos reutilizar parte de los datos que ya tenemos especificados en (data.sql).
- No afecta al trabajo diario de desarrollo y pruebas de la aplicación
- Es simple, sólo hay que redactar las secuencias SQL

Inconvenientes:

- Tenemos que buscar nosotros listas con datos reales si queremos mantener la coherencia
- La cantidad de datos es mucho menor
- Casos de uso no realistas
- Mucho más lento

Justificación de la solución adoptada

Como consideramos que es fundamental poder aportar casos de uso reales, que reflejen cómo sería el verdadero desempeño del sistema en una situación de este tipo hemos seleccionado la alternativa 1.a.

4.2. Decisión 2: Eliminación de los vehículos

Descripción del problema:

Inicialmente planteamos la eliminación de vehículos del sistema dado que, por temas de antigüedad, desgaste o simplemente dar paso a nuevos modelos, el administrador pudiese eliminarlos. El problema que conlleva hacer esto, es que estaríamos eliminando datos que podrían ser de utilidad en un futuro, cómo por ejemplo un precio de referencia para otros modelos del mismo año, qué incidencias eran frecuentes o la demanda del mismo para su posible reintroducción.

Alternativas de solución evaluadas:

Alternativa 2.a: Retirar los vehículos, cambiándoles su disponibilidad a “No disponible”, permitiendo al administrador, dar de alta o de baja un vehículo cuando lo considerase necesario.

Ventajas:

- No se pierde ningún dato que pueda ser útil de cara a un futuro
- Podemos reutilizar parte del trabajo empleado en otros controladores

Inconvenientes:

- Sobrecarga aún más la vista de vehículos, añadiendo una pestaña más y botones, dificultando posibles cambios que sean necesarios
- Es algo complejo, pues los controladores redirigen a la misma página y han de no interactuar a la vez sobre los mismos datos

Alternativa 2.b: Eliminar directamente los vehículos del sistema, cuando el administrador lo considere necesario

Ventajas:

- Muy simple, no interviene con otras funcionalidades
- Complejidad prácticamente nula

Inconvenientes:

- No contempla casos en los que la retirada del vehículo sea solo temporal, cómo por ejemplo un mantenimiento
- Si se necesita en un futuro ese vehículo sería necesario su reintroducción en el sistema

Justificación de la solución adoptada

Consideramos que es necesario no eliminar los vehículos pues pueden ser de utilidad en un futuro y permiten mantener un registro, por lo que tomamos la alternativa 2.a.

4.3. Decisión 3: Seguro cliente

Descripción del problema:

En un principio contemplamos añadir contratos de venta y de alquiler como entidades y en ese contrato de alquiler iría el seguro del cliente. Eliminamos esas dos entidades tras una revisión del proyecto y a la hora de añadir el seguro del cliente nos faltaba una entidad.

Alternativas de solución evaluadas:

Alternativa 3.a: Añadir de nuevo la entidad contrato de alquiler, que iría relacionada con el seguro del cliente.

Ventajas:

- Las historias de usuario varían poco y la planificación sigue igual.

Inconvenientes:

- Añadir una nueva entidad a estas alturas conlleva mucho tiempo y genera conflictos.

Alternativa 3.b: Plantear todo lo relacionado con el seguro del cliente de otra forma. De esta manera el seguro del cliente va asociado directamente al vehículo y lo crea, edita o elimina el gestor. El cliente puede alquilar un vehículo con un seguro de cliente solo si se ha asociado un seguro a dicho vehículo.

Ventajas:

- La modificación de las historias de usuario es relativamente pequeña.

Inconvenientes:

- Todo el proceso relacionado con el seguro del cliente es un poco más complejo.

Justificación de la solución adoptada

Debido al tiempo y los conflictos que conllevaría añadir una nueva entidad hemos optado por la alternativa 3.b.

4.4. Decisión 4: Eliminación de ofertas

Descripción del problema:

A la hora de programar la parte relacionada con las ofertas nos surgió la duda de si era mejor eliminar las ofertas o simplemente ocultarlas o archivarlas.

Alternativa 4.a: Eliminar las ofertas.

Ventajas:

-Si hay alguna oferta errónea u obsoleta podría ser eliminada de por vida

Inconvenientes:

-No existiría un historial de ofertas ni la posibilidad de reutilizar una misma oferta en diferentes momentos.

Alternativa 4.b: Ocultar o archivar las ofertas.

Ventajas:

-Existiría un historial de ofertas que nos permitiría consultar ofertas pasadas y podría utilizarse una misma oferta en varias ocasiones, puesto que las ofertas archivadas u ocultas se podrían mostrar de nuevo.

Inconvenientes:

-Si alguna oferta es errónea o queda obsoleta no podría borrarse.

Justificación de la solución adoptada

Hemos optado por la alternativa 4.a porque pensamos que desde el punto de vista de la funcionalidad se deben de borrar las ofertas erróneas u obsoletas, dejando lo más claro y limpio posibles que ofertas estás activas actualmente y cuáles no.

4.5. Decisión 5: Asignación de vehículos a una oferta

Descripción del problema:

Con relación al apartado de ofertas, se nos planteó la duda de si una oferta se asociaría a un solo vehículo, o, por el contrario, esta podía estar asociada a más de uno.

Alternativa 5.a: Una oferta está asociada a un solo vehículo.

Ventajas:

- Independencia de las ofertas. Cada oferta se corresponde con un solo vehículo
- Se mantienen las relaciones planteadas en el modelado

Inconvenientes:

- Puede haber varias ofertas iguales asociadas cada una a un vehículo diferente

Alternativa 5.b: Una oferta puede estar asociada a varios vehículos.

Ventajas:

- Disminuye la cantidad de ofertas, ya que una oferta puede referirse a varios vehículos. No es necesario crear siempre una nueva oferta

Inconvenientes:

- Puede generar confusión, ya que cuando miramos una oferta temenos que fijarnos en todos los vehículos a los que está asignada
- Es necesario cambiar algunas relaciones para poder llevar a cabo la alternativa

Justificación de la solución adoptada

Pensamos que es más útil poder asignar una misma oferta a varios vehículos y evitar de esta manera las ofertas repetidas. Además, los cambios en las relaciones no suponían grandes conflictos ni mucho tiempo, por lo que hemos optado por la opción 5.b.

4.6. Decisión 6: Eliminación de entidades de contratos

Descripción del problema:

Cuando teníamos todo el proyecto modelado nos dimos cuenta de la existencia de dos entidades que carecían de atributos y representaban los contratos de alquiler y venta. Tras una revisión del proyecto llegamos a la conclusión de que los contratos eran archivos más que entidades y necesitábamos encontrar una forma de solucionar este problema.

Alternativas de solución evaluadas:

Alternativa 6.a: Eliminar las dos entidades y considerar contratos como archivos.

Ventajas:

- Eliminamos entidades sin atributos que carecen de sentido

Inconvenientes:

- Llevar a cabo los cambios en el resto del modelado

Alternativa 6.b: Buscar atributos para ambas entidades.

Ventajas:

- No hay que modificar nada del modelado

Inconvenientes:

- Tener entidades poco útiles con atributos innecesarios

Justificación de la solución adoptada

Hemos optado por la alternativa 6.a porque de esta manera eliminábamos entidades que resultaban poco útiles en el proyecto.

4.7. Decisión 8: Respuesta de error de la API

Descripción del problema:

En caso de no estar bien construida la llamada de la API por algún fallo en el body de la misma, construir una respuesta de "bad request" personalizada donde contenga información de los errores.

Alternativas de solución evaluadas:

Alternativa 8.a: Construir la bad request con el cuerpo vacío, simplemente con el código HTTP 400.

Ventajas:

- Implementación sencilla.

Inconvenientes:

- Respuesta sencilla, no aporta mucha información de lo que fue mal.
- Una vez llamada a la API con errores en el body de construcción, es muy complicado mostrar en la vista de representación de la API los errores cometidos asociados a los parámetros determinados.

Alternativa 8.b: Construir la bad request con un cuerpo personalizado.

Ventajas:

- Si se llama a la API con errores en el body, el mensaje de respuesta será una bad request acompañada en el cuerpo de los parámetros donde hay errores.
- Implementación muy sencilla a la hora de mostrar los errores en la vista.

Inconvenientes:

- Implementación compleja, es necesario crear una clase que modele un json con el cuerpo de error conteniendo las keys principales para mostrar la información de error.

Justificación de la solución adoptada

Hemos optado por la alternativa 8.b puesto que el mensaje devuelto por la API en caso de una petición incorrecta es mucho más rico en cuanto a información y facilita mucho a la hora de mostrar los errores en la vista.

4.8. Decisión 9: Implementación de buscadores

Descripción del problema:

Queríamos implementar y también debíamos (por lo descrito en una historia de usuario) un sistema de búsqueda de algunas entidades. Estas entidades al final han sido Vehículos, Cliente y Concesionario.

Alternativas de solución evaluadas:

Alternativa 8.a: Hacer queries directamente a la base de datos desde el repositorio.

Ventajas:

- Más simple de implementar

Inconvenientes:

- No es posible implementar búsquedas por distintos atributos, solo por uno.

Alternativa 8.b: Usar Hibernate Search para implementar las búsquedas.

Ventajas:

- Nos permite buscar por distintos atributos de una clase. Por ejemplo, para Concesionario se puede buscar por nombre, localidad, provincia y país.

Inconvenientes:

- Es un poco más complicado de entender y de implementar.

Justificación de la solución adoptada

Hemos optado por la alternativa 8.b puesto que buscar datos por distintos atributos mejora la experiencia de usuario. Además, este método de búsqueda indexa cada atributo que se puede buscar lo que permite hacer búsquedas parecidas a las de google. Por ejemplo, un concesionario llamado Veyser auto se mostrará tanto si buscas [V,Ve,… Veyser] como si buscas [a,au,..,auto].

4.9. Decisión 10: Implementación de paginación

Descripción del problema:

Queríamos implementar un sistema de paginación de algunas entidades. Estas entidades al final han sido Vehículos, Cliente, Concesionario, Oferta y Venta.

Alternativas de solución evaluadas:

Alternativa 9.a: No implementar la paginación y limitar el volumen de datos debido a que mostrar una gran cantidad de datos en una misma página sería incómodo para el usuario.

Ventajas:

- No es necesario implementar nada
- Ahorro de tiempo

Inconvenientes:

- Limitar volumen de datos
- No es agradable visualmente
- Incómodo para el usuario

Alternativa 9.b: Implementar la paginación en aquellas entidades en las que hubiese una gran cantidad de datos como por ejemplo la entidad Vehículo.

Ventajas:

- Volumen de datos ilimitado
- Agradable visualmente
- Cómodo para el usuario

Inconvenientes:

- Implementar la nueva funcionalidad en las diferentes entidades añadiendo nuevos métodos a los repositorios, servicios y controladores
- Realizar test de los métodos adicionales
- Conlleva una gran cantidad de tiempo

Justificación de la solución adoptada

Hemos optado por la alternativa 9.b puesto que paginar los datos mejora la experiencia de usuario. Además, permite tener un volumen de datos mayor y es más sencillo acceder a datos que se encuentran al final de la tabla.

5. Logs

Hemos implementado varios logs a lo largo del desarrollo de la aplicación asociado a su nivel de importancia sintáctico correspondiente. Antes de proceder a describir cada uno de ellos, es interesante comentar que hemos implementado la política de file-rolling de tal forma que se va ir archivando los logs de nivel info y cuando supere 10Mb el fichero va a ser sustituido por uno nuevo y este va a ser almacenado en /dp1-2020-g1-05/logs/archived. Toda esta configuración está presente en el fichero logback.xml, con ruta: /dp1-2020-g1-05/src/main/resources/logback.xml.

Criterios seguidos para clasificar los logs a su nivel correspondiente:

- Info: logs para uso analítico o puramente aclarativo, se pueden aprovechar para sacar más información y detalles sobre diversas partes de la aplicación.
- Warn: logs que muestran algún tipo de aviso sobre un suceso que no ha ido como tenía que haber ido, pero permite seguir con la funcionalidad principal del objeto.
- Error: logs que muestran un fallo grave en la aplicación impidiendo continuar con la funcionalidad.

Los logs presentes se encuentran:

- AlquilerController.java:
  - initAlquilarVehiculo(int, Map<String,Object>): contiene 2 logs de nivel info, que registran que la operación de alquilar no se ha podido llevar a cabo debido a que el vehículo esta ya alquilado o está en revisión. También se encuentra un log de nivel warn, que indica que el cliente es conflictivo y no se puede realizar el alquiler.
  - ProcessAlquilarVehiculo(int,Alquiler, BindingResult): contiene un log de nivel warn indicando que no se ha podido completar la operación de alquiler y otro de nivel info para mostrar que el vehículo con su id correspondiente ha sido alquilado.
- ClienteController.java:
  - FindAll(Map<String,Object>, ModelMap): contiene un log de nivel warn para indicar que algo ha ido mal al cargar todos los clientes.
- ConcesionarioController.java:
  - SearchConcesionarios(String, ModelMap): contiene un log de nivel info almacenando la búsqueda realizada, interesante desde el punto de vista analítico para ver qué búsqueda se realizan respecto a los concesionarios.
- CrashController.java
  - TriggerException(): contiene un log de nivel error para registrar que se ha indicado un grave error en la aplicación.
- OfertaController.java
  - ShowOferta(int, ModelMap): contiene un log de nivel info para registrar que se ha visto los detalles de una determinada oferta, interesante desde el punto de vista analítico para conocer más detalles sobre qué ofertas interesan más.
- ReservasController.java
  - ProcessReservarVehiculo(int, String, Reserva, BindingResult): contiene un log de nivel info para registrar los vehículos reservados y así tener más detalles sobre los vehículos que más se reservan.
- UserController.java
  - ProcessCreationForm(Cliente,BindingResult): contiene un log de nivel warn para registrar los errores en el registo de un nuevo cliente.
- VehiculosController.java

    - FindAll(Map<String,Object>,ModelMap): contiene un log de nivel warn para indicar que algo ha ocurrido mal ya que no se ha obtenido ningún vehículo de la base de datos.
    - ShowVehiculo(int): contiene un log de nivel info para registrar que vehículo se consulta más sus detalles.
    - DeleteVehiculo(int,Map<String,Object>): con tiene un log de nivel info para registrar que el vehículo ha pasado al estado no disponible
    - SearchVehiculos(String,ModelMap): contiene un log de nivel info para registrar qué búsquedas asociadas a los vehículos son las que más se realizan.
- VentaController.java
    - ComprarVehiculo(int, Map<String, Object>): contiene doslogs de nivel info para registrar que no se ha podido comprar ya que el vehículo ya está vendido o bien está en revisión y otro log de nivel info también para registrar que el vehículo con su ID correspondiente ha sido registrado.
- ClienteService.java
    - SaveCliente(Cliente): contiene un log de nivel info para registrar el ingreso de un nuevo usuario.
- CustomErrorController.java
    - HandlerError(HttpServletRequest): contiene logs de nivel warn para cada tipo de error que gestiona este método.

6. Funcionalidades A+:

API REST

Descripción de la funcionalidad:

Hemos añadido un servicio REST a nuestra aplicación, a los recursos de Oferta, Concesionario y Alquiler.

Clases o paquetes creados

En el paquete de controlador, com/springframework/samples/madaja/web, hemos creado:

- AlquilerControllerAPI.java
- ConcesionarioControllerAPI.java
- VehiculosControllerAPI.java

En el paquete de útil, com\springframework\samples\madaja\util, hemos creado:

- APIerror.java

En el paquete de controlador, com/springframework/samples/madaja/web, hemos añadido los siguientes métodos a los siguientes controladores:

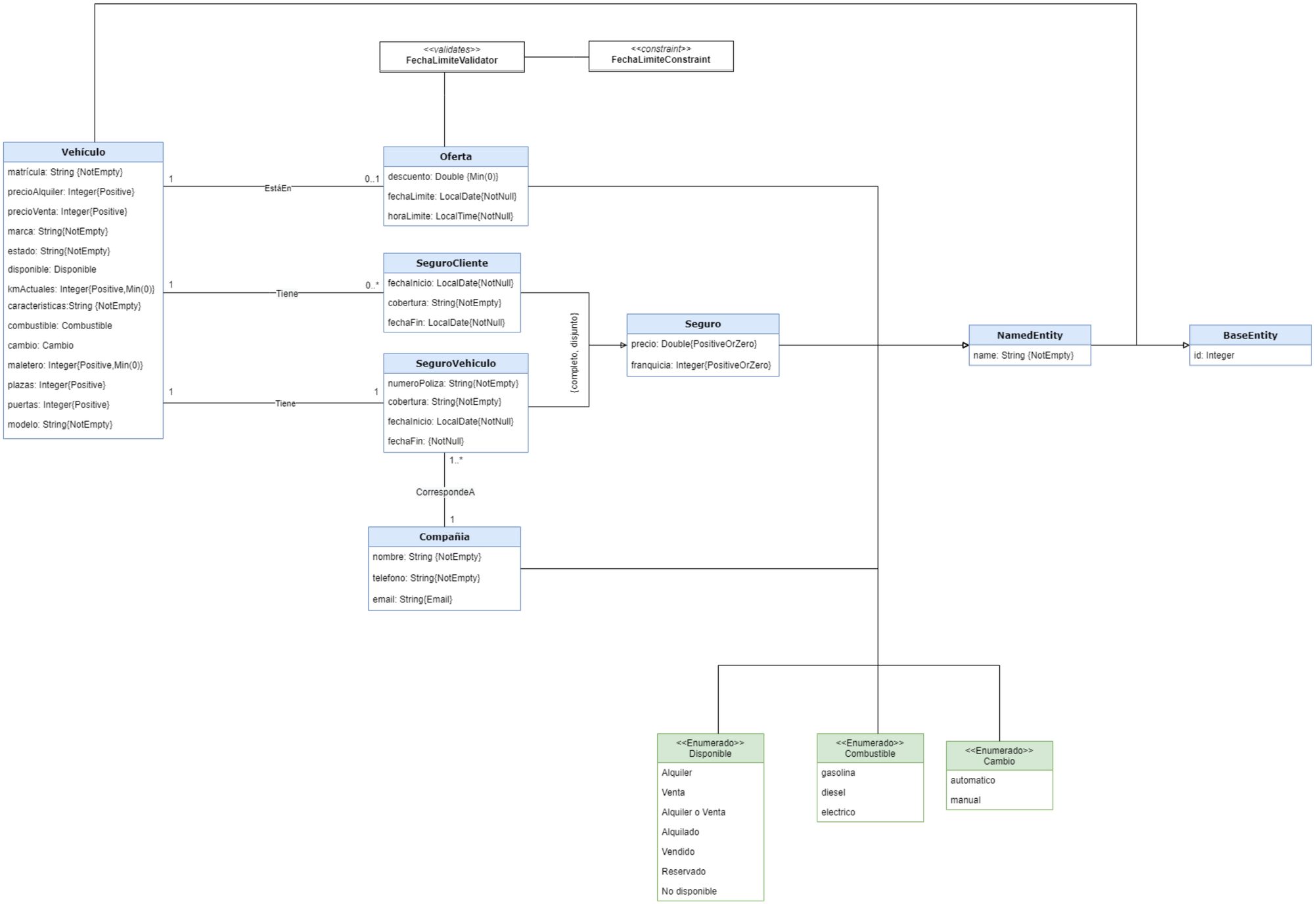
- OfertaController.java:
    - showOfertasListAPI()
    - showOfertaAPI()
    - creationOfertaAPI()
    - updateOfertaAPI()
    - deleteOfertaAPI()
- AlquilerController.java:

    - showMisAlquileresListAPI()
- ConcesionarioController.java
    - showConcesionarioListAPI()
    - showConcesionarioAPI()
    -

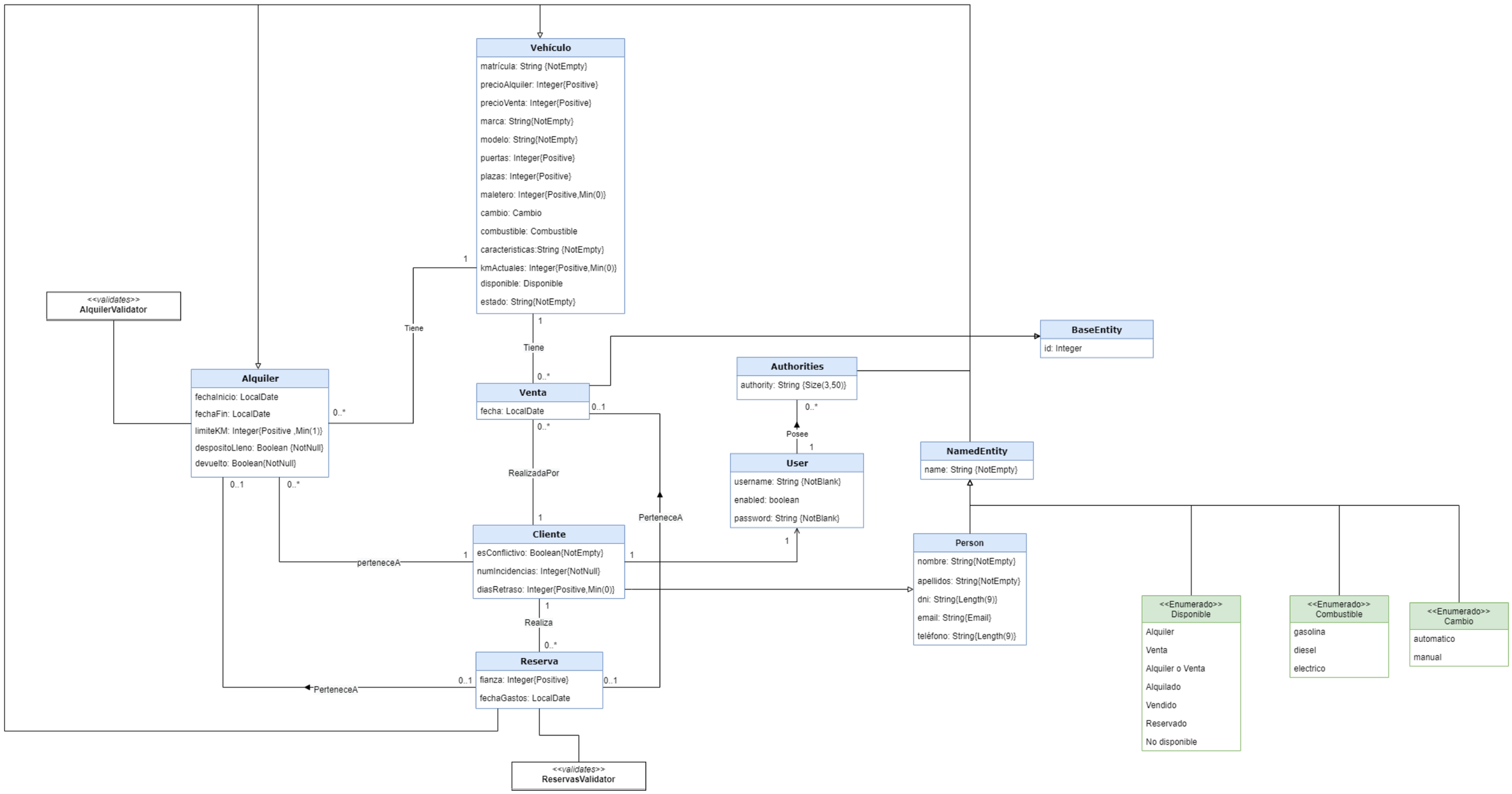
Se han creado nuevas vistas y scripts para realizar las llamadas AJAX:

- src\main\resources\static\resources\js:
    - concesionarioDetails.js
    - createOfertaAPI.js
    - mostrarConcesionarios.js
    - mostrarMisAlquileres.js
    - mostrarOfertas.js
    - ofertaDetails.js
    - updateOfertaAPI.js
- src\main\webapp\WEB-INF\jsp\oferta:
    - createOfertaFormAPI.jsp
    - mostrarOfertasAPI.jsp
    - ofertaDetailsAPI.jsp
    - updateOfertaAPI.jsp
- src\main\webapp\WEB-INF\jsp\concesionario:
    - concesionarioDetails.jsp
    - mostrarConcesionariosAPI.jsp
- src\main\webapp\WEB-INF\jsp\alquiler
    - mostrarMisAlquileresAPI.jsp

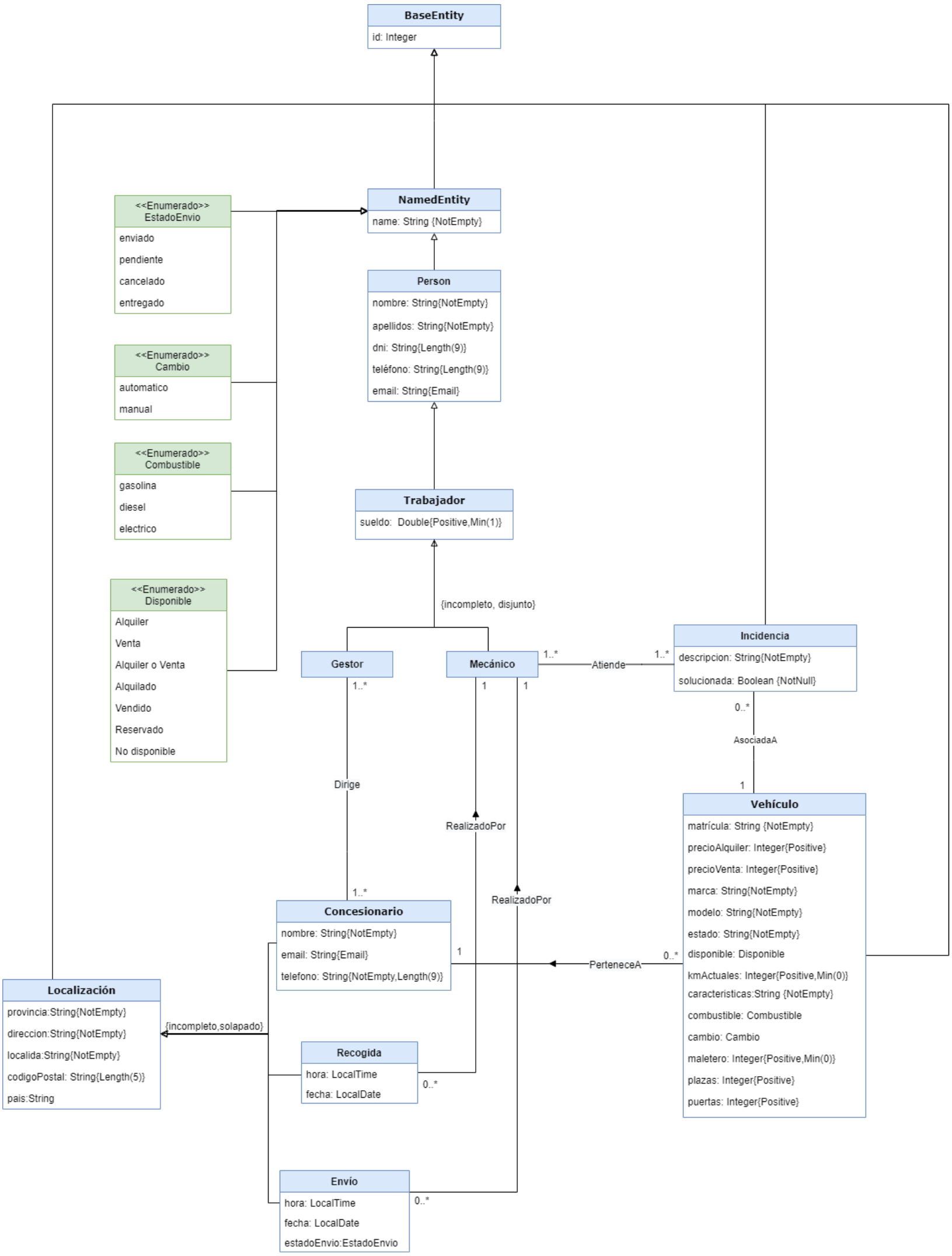
Anexo I. Diagrama de Diseño/Dominio-Parte I: Seguros y oferta



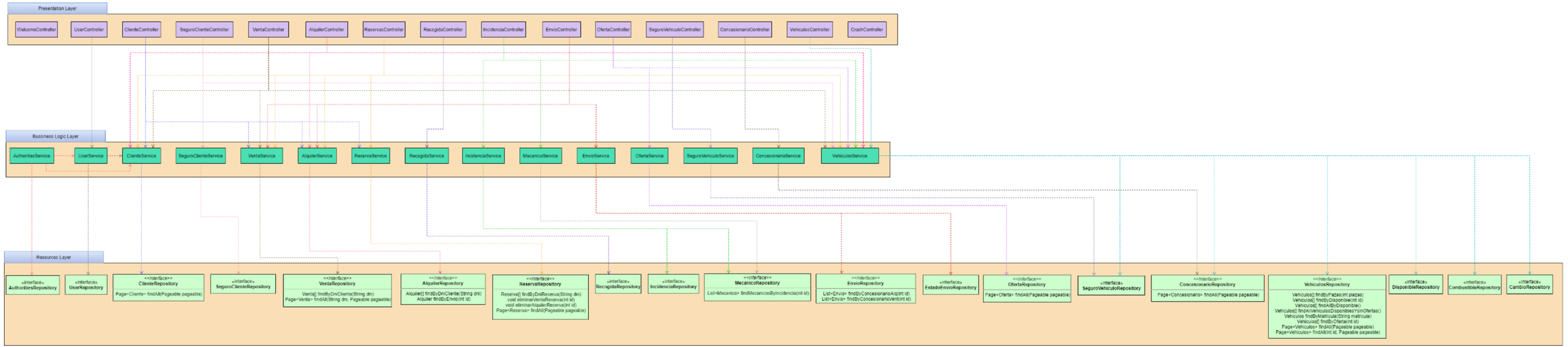
Anexo II. Diagrama de Diseño/Dominio-Parte II: Alquiler, Reserva, Venta y Cliente



Anexo III. Diagrama de Diseño/Dominio-Parte III-Localización,Concesionarios, Incidencias y Trabajadores



Anexo IV. Diagrama de Capas



Anexo V. Diagrama de Capas-Parte principal Vehículos